University of California Santa Cruz
Department of Computer Engineering
Lab Experiment Report # 4

Counters

Author: Kyle Jeffrey
Lab Partner: NA
Due Date: 2/15/18

Objective

Create a 16-bit counter that holds a number when no buttons are pressed, counts continuously, loads a value, or counts up/down incrementally each time the respective button is pressed. This system requires an edge trigger, a 4-bit counter, a ring counter, a selector, and reuses the 7-segment display logic.

Kyle Jeffrey
Prof Schlag
CE 100
1/18/18

Counter Design:

Using the included flip-flop module provided by verilog, make a counter with inputs:

- the system clock,
- Up(increment)
- Dw(decrement)
- LD(load control)
- the 4-bit vector D that will be loaded on the clock edge if LD is high.

And the Outputs:

- a 4-bit bus Q which is the current value held by the counter,
- the signal UTC (Up Terminal Count) which is 1 only when the counter is at 1111, and
- the signal DTC (Down Terminal Count) which is 1 only when the counter is at 0000.

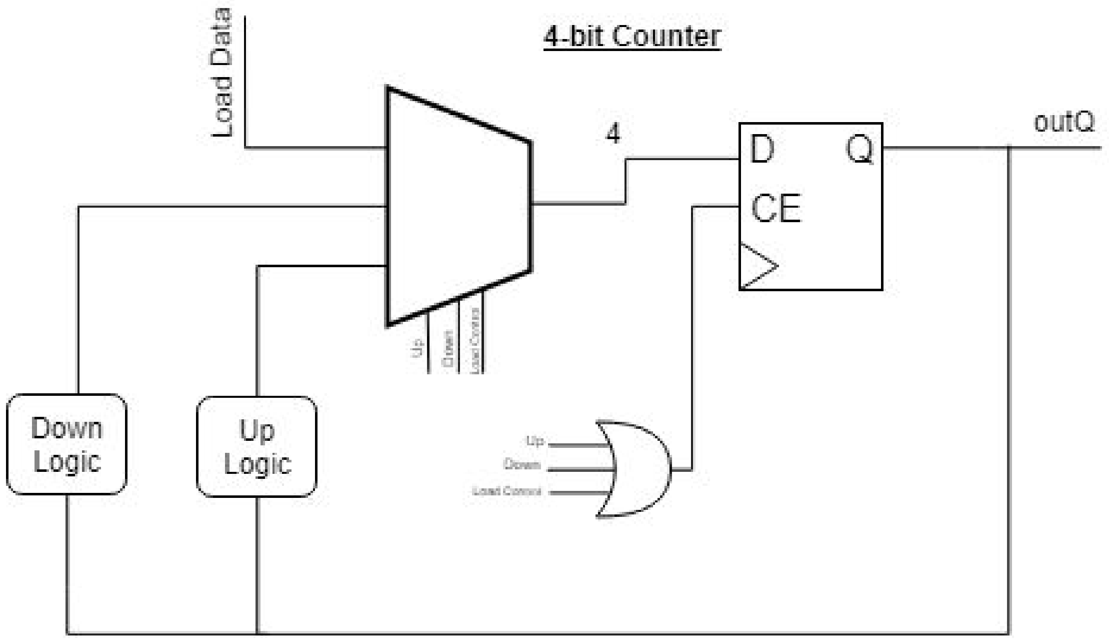
The counter should go all the way around when reaching the lowest or highest numbers, specifically from 0000 to FFFF and vice versa.

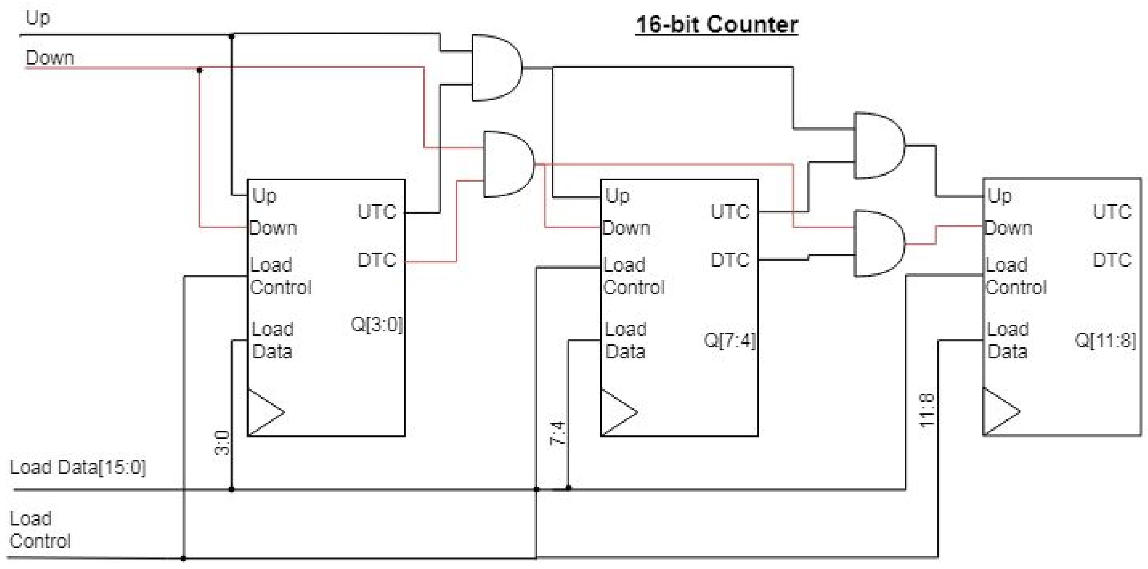
Method:

4-bit counter: Using a state truth table to determine how the flip flops go from the present state to the next state, equations are determined included in the results section. The counter must be able to go up, down, and load a 4 bit value. A mux* is placed in front of the data ports of the 4 flip flops holding the output values. The mux is loaded with logic for an up operation*, logic for a down operation*, and the 4 bits of input data. The only output logic here is DTC for downwards terminal count and UTC for upper terminal count, for when 0000 and FFFF is hit, using a 4 input AND gate.

16-bit counter: Putting four 4-bit counters rippled in ascending of bits from 0 to 16 together gave the 16-bit counter. The UTC output and DTC output were connected to each counter following it in ascending bit significance. A shell module for the 16 bit counter was made, with a 1-bit up, down and load control input, and a 16-bit load input, as the control inputs that would come from the FPGA board. DTC and UTC was also included in the 16-bit version of the counter, for possible bigger counters in the future.

RESULTS:
These were both tested by hardcoding up's and down's. The boundary cases of 0 to F were important for the 4 bit and from FFFF to 0000 for the 16-bit.

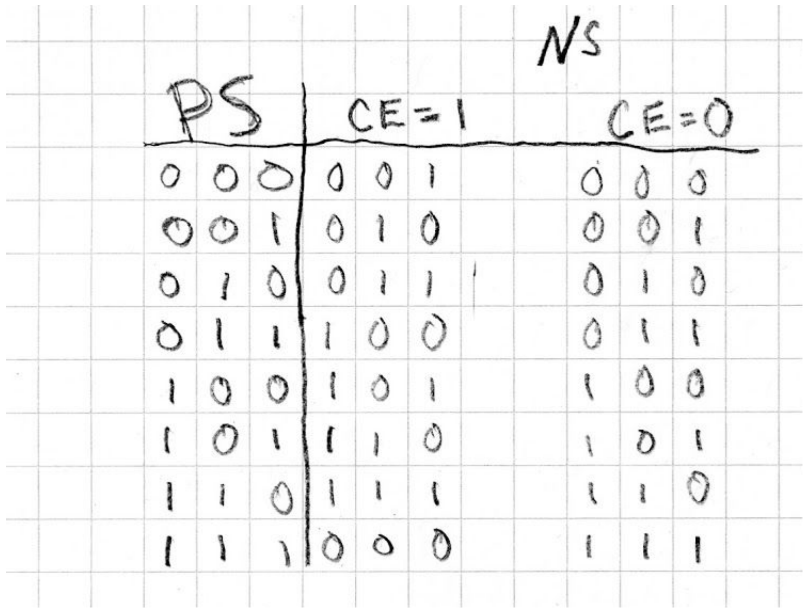




The diagram is missing one last 4-bit counter on the end for space purposes

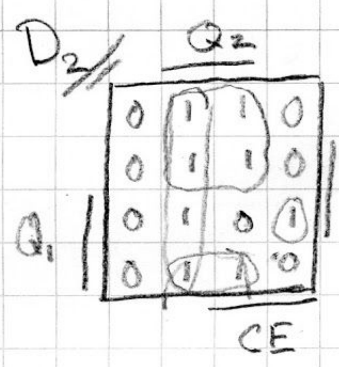
Kyle Jeffrey
Prof Schlag
CE 100
1/18/18

UP LOGIC

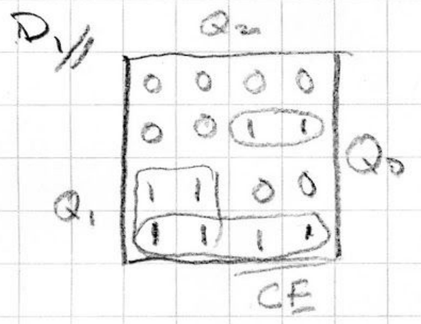


NS

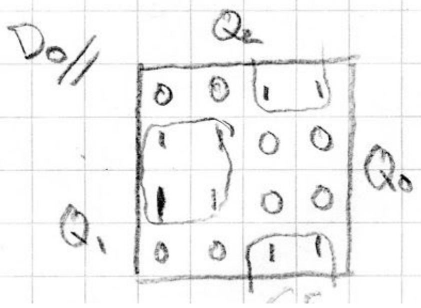
PS			CE=1			CE=0		
0	0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
1	0	0	1	0	1	1	0	0
1	0	1	1	1	0	1	0	1
1	1	0	1	1	1	1	1	0
1	1	1	0	0	0	1	1	1



$$D_2 = Q_2 \oplus CEQ_1Q_0$$



$$D_1 = CE\,\overline{Q_1}\,Q_0 + \overline{CE}\,Q_1 + Q_1\overline{Q_0}$$
$$= Q_1(\overline{CE\,Q_0}) + \overline{Q_1}(CE\,Q_0)$$
$$= Q_1 \oplus CE\,Q_0$$



$$D_0 = CE\,\overline{Q_0} + \overline{CE}\,Q_0$$
$$= CE \oplus Q_0$$

The equations for the input data follow this algorithmic trend such that D_i = Q_i xor CE & Q_i & Q_{i-1}…. The down input data happen to be very similar and end up being D_i = Q_i xor CE & !Q_i & !Q_{i-1}…. Which was given to us in Lecture.

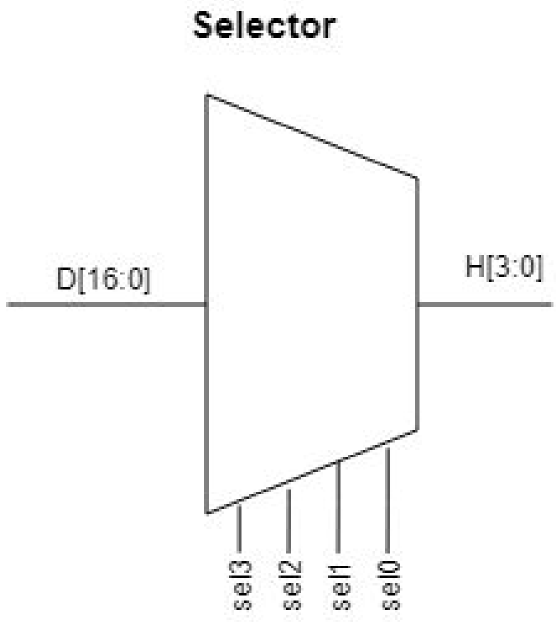
Selector:

The selector determines which 4-bits will be associated with which hex display. With the input coming from the ring counter, the selector gives the bits while the ring counter determines which hex digit to display, which requires 4 bits. The selected four bits will be sent to the display.

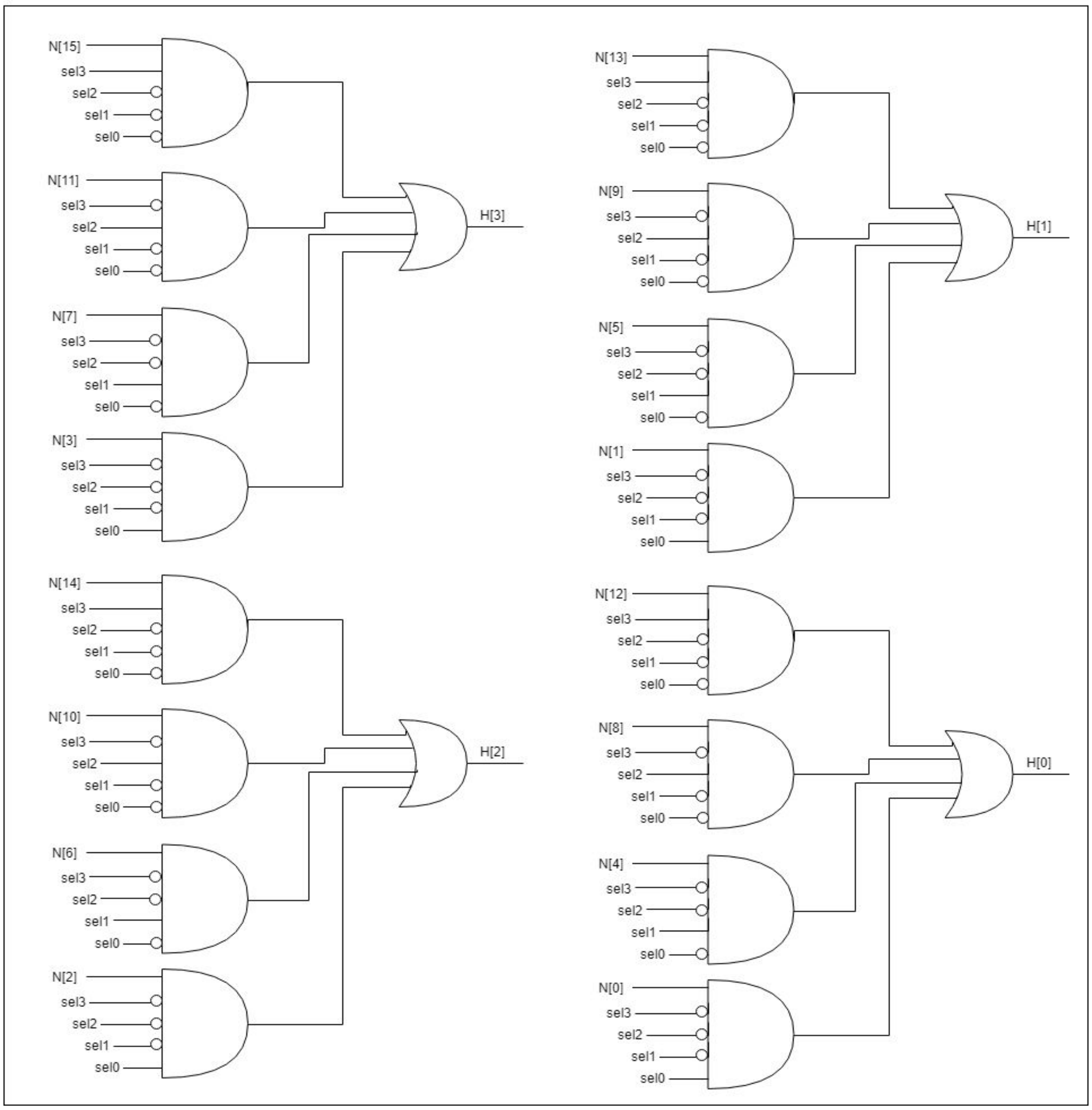
H is x[15:12]	when sel=(1000)
H is x[11:8]	when sel=(0100)
H is x[7:4]	when sel=(0010)
H is x[3:0]	when sel=(0001)

Method:
The selector behaves similar to a 12 to 4 mux. Four selector inputs selector which four bits will be used. Thinking about each group of four as one of the hex digits to be displayed to the board provides a good way of thinking about why this is needed. Because only 7 inputs control the four hex displays, we must cycle through each display rapidly at a high frequency so the eye can't detect that we're actually only displaying 1 hex digit at any given time.

Results:



Selector



In simulation, the selector just needed to select the asked for section of 4-bits handed to it by a Load and the ring counter.

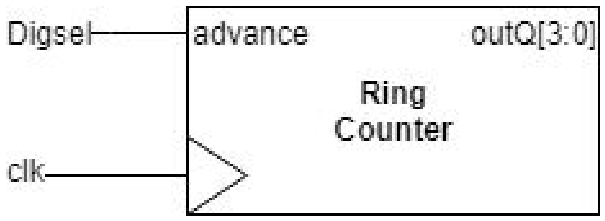
Ring Counter:

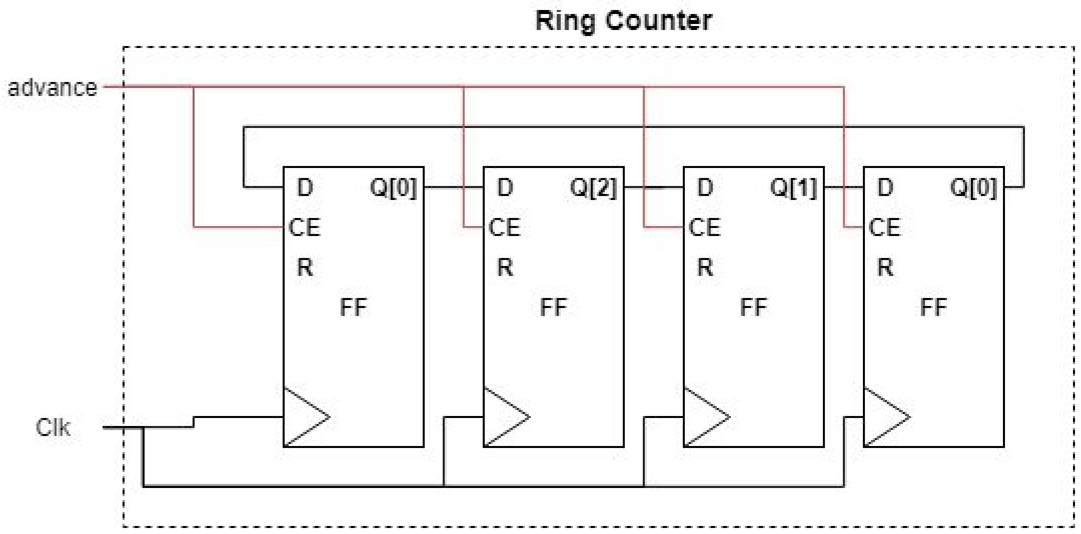
The ring counter only ever has a single 1 in it at anytime. As a 4-bit vector, each index corresponds to an anode of the hex display. The ring counter should be thought of as a device that cycles through the displays to activate them individually at a high frequency giving the illusion that all 16-bits are displayed at one time.

Method:

The ring counter is comprised of four flip flops in series, connected output to input in a "ring". This means that it cycles from 0001 all the way back to 1000. An advance input is connected to a clock that advances the 1 through the ring.

Results:





All the resets should be set to 0.

The simulation just needed to move a 1 forward with each advance input, no edge detection was required.

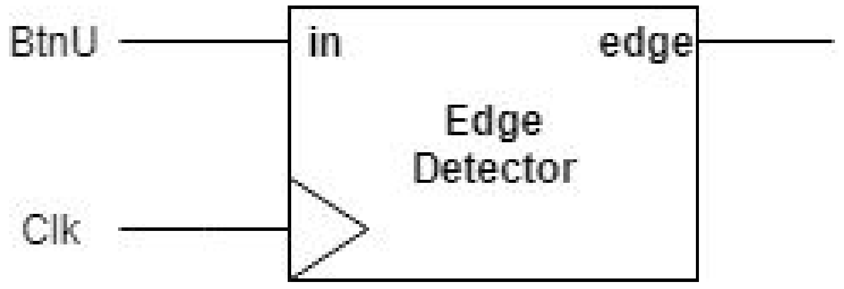
Edge Detector:

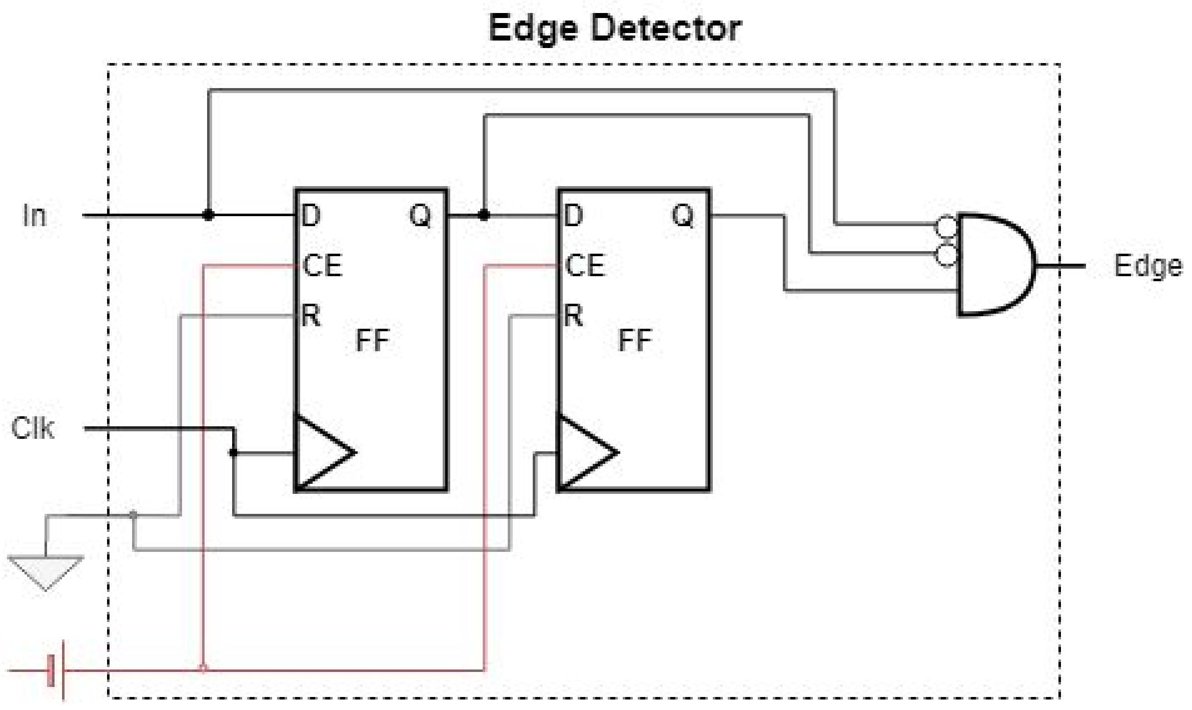
The edge detector gives us an up and down button that don't count continuously. The module detects a value going from low to high making it a positive edge trigger.

Method:

Using two flip flops to connect the last to values, an input bit goes into the first flip flops data and the first output is connected to the second input. This stores the two most recent input values. To remove shaky inputs, the output is an and gate that detects two 0's followed by a single 1.

Results:





btnC Stopper:

At FFFC the continuous up should stop. BtnC should still be able to count up and btnD should be able to count down.

Method:

FFFC in binary equals 1111111111111100. So when the 14 MSB are not equal to 1 then the counter should go up continuously.

Results:

To get a high value for every number below FFFC, NAND the 14 MSB so every value not all 1's is true. AND this with btnC to get the continuous up input with a stop in it.

Top Level Design:

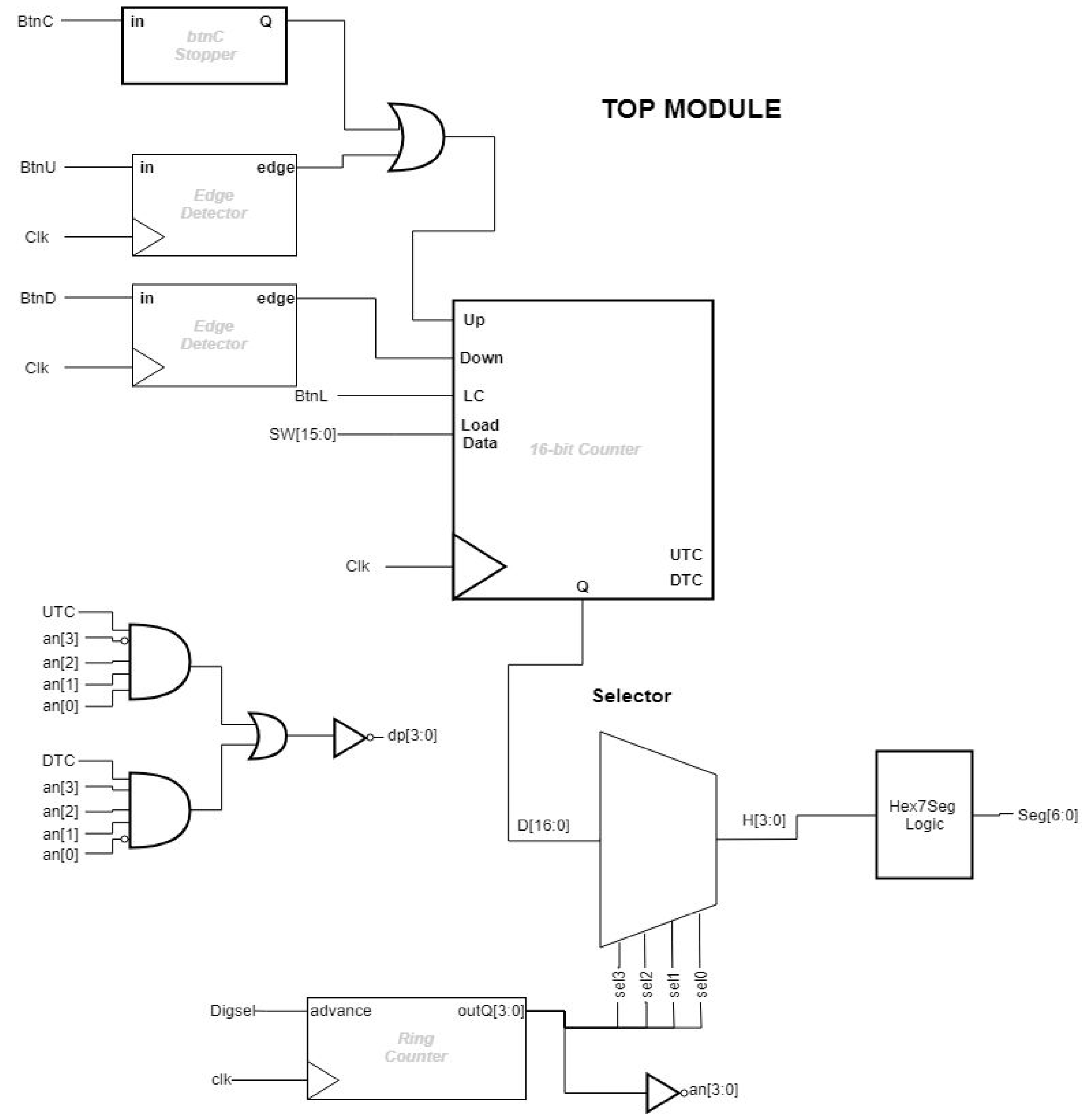
The top module requires a button that continuously counts up, a button that incrementally counts up or down with each press of the button, and displays 16 bits. The continuous up will stop from FFFC to FFFF. A period will be placed on the left most hex display when FFFF is reached and a period will be displayed on the right most display when 0000 is the current value. 16 switches will control a loadable 16 bit value imported with a button press. T

Method:

Edge Detectors are placed in front of btnC and btnU. The btnStopper module is placed in front of btnC to hard stop the continuous up when FFFC is reached, and this output is OR'd with the btnU and connected to the up input of the 16-bit counter. LoadData and LoadControl are connected to the switches and btnL directly. The output of the counter goes into the selector which is controlled by the ring counter which advances through the activating the displays at the pace of the digsel clock pulse provided by the labs clock module provided by the lab. This puts the bits in line with which hex display is on, because the ring counter output also controls the anodes of the hex display.

Simulating the top module just meant switching through up and down button inputs to assure that the counter responded accordingly. It's very difficult to access whether the hex display is displaying the correct digits so, all of the data checking is on the output of the counter. The most important tests were determining that there was no continuous counting by btnC and btnD and that the counter stopped at FFFC and flipped all the way around from FFFF to 0000.

Results:


TOP MODULE
BtnC
in
Q
btnC Stopper
BtnU
in
edge
Edge Detector
Clk
BtnD
in
edge
Edge Detector
Clk
Up
Down
BtnL
LC
SW[15:0]
Load Data
16-bit Counter
Clk
UTC
DTC
Q
UTC
an[3]
an[2]
an[1]
an[0]
DTC
an[3]
an[2]
an[1]
an[0]
dp[3:0]
Selector
D[16:0]
H[3:0]
Hex7Seg Logic
Seg[6:0]
sel3
sel2
sel1
sel0
Digsel
advance
outQ[3:0]
Ring Counter
clk
an[3:0]

Kyle Jeffrey
Prof Schlag
CE 100
1/18/18

Question 1: Hold down btnR. What happens?

Answer: Only the right most hex display is turned on and it's set to 0. This is due to the ring counter which has the first hex display initialized to 1 with all others set to 0.

Question 2: The module **lab4_clks** has another output port named **fastclk**. Connect all the clock inputs (**Edge Detector**, **Counter**, **Ring Counter** and your synchronizer (if you wisely used one for btnU and btnD) to the port **fastclk** instead of **clk**. Leave the advance input of the ring counter as before (connected to the **digsel** output of **lab4_clks**). Implement and Configure. What happens?

Answer: Nothing, the modules work the same as before.

Conclusion

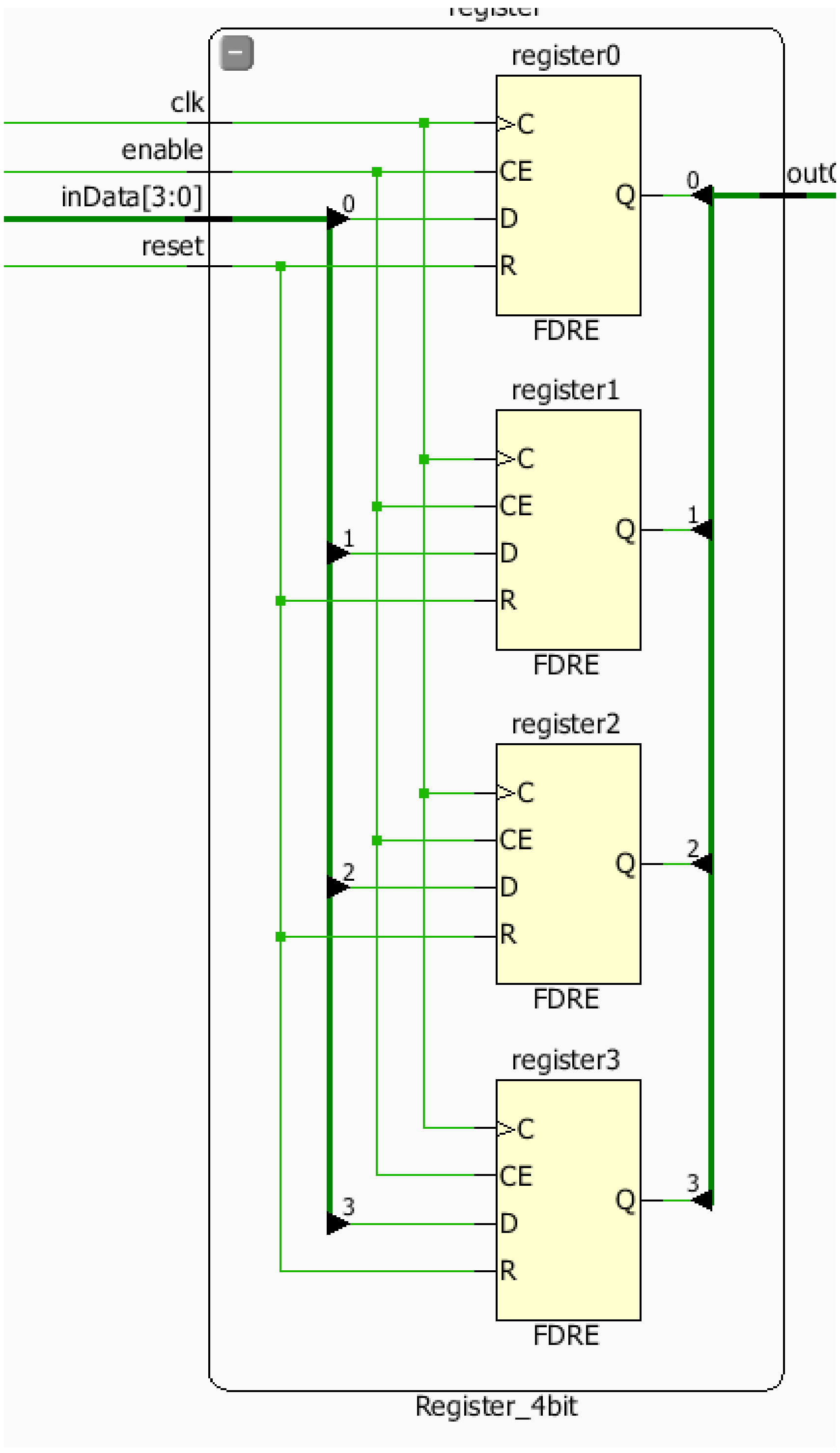
This lab began the classes introduction into combinational logic. Learning to use state truth tables slowly introduced us into the idea of states. This also introduced displaying digits on multiple displays with only one hex input. I started early when making this lab and hit a roadblock when attempting to use adders for up and down instead of learning counter design. If that taught me anything, it's to slow down and to be welcome to learning new strategies for solving problems rather than relying on old tricks. Using counters is much simpler than using two full adders to add or subtract 1. All of the modules seem to have a fairly streamlined designed though., so I don't believe any corrections needed to be made.

```
`timescale 1ns / 1ps


module countUD4L(
    input clk, up, down, loadControl,
    input [3:0] loadData,
    output upperLimit, lowerLimit,
    output [3:0] outQ
    );
    wire [3:0] q, D, dw, u;


    assign upperLimit = q[3] & q[2] & q[1] & q[0];
    assign lowerLimit = ~q[3] & ~q[2] & ~q[1] & ~q[0];


    // Up logic goes into selector
    assign u[0] = q[0] ^ up;
    assign u[1] = q[1] ^ (up & q[0]);
    assign u[2] = q[2] ^ (up & q[0] & q[1]);
    assign u[3] = q[3] ^ (up & q[0] & q[1] & q[2]);

       //down logic goes into selector
     assign dw[0] = q[0] ^ down;
     assign dw[1] = q[1] ^ (down & ~q[0]);
     assign dw[2] = q[2] ^ (down & ~q[0] & ~q[1]);
     assign dw[3] = q[3] ^ (down & ~q[0] & ~q[1] & ~q[2]);

     // selector determines which input goes into 4 bit register based on what button
pressed
     // Mux is controlled by selector determing add or subtract.
     // Becuase mux is always connected directly to the register, an enable must be
tied to everytime an operation is pressed.
     // add 001, sub 010, load 100
     m3_1x4 op(.in0(u), .in1(dw), .in2(loadData), .o(D), .sel2(loadControl),
.sel1(down), .sel0(up));

     //4 bit register
     // When mux has no inputs selected, it is in invalid state. This means our
enable should only be true
     // when pressing a button so the register doesnt enter invalid state
    Register_4bit register  (.reset(1'b0), .clk(clk), .outQ(q), .inData(D),
.enable(up | down | loadControl));

    //output comes from output of register
    assign outQ = q;
```

```
endmodule
```

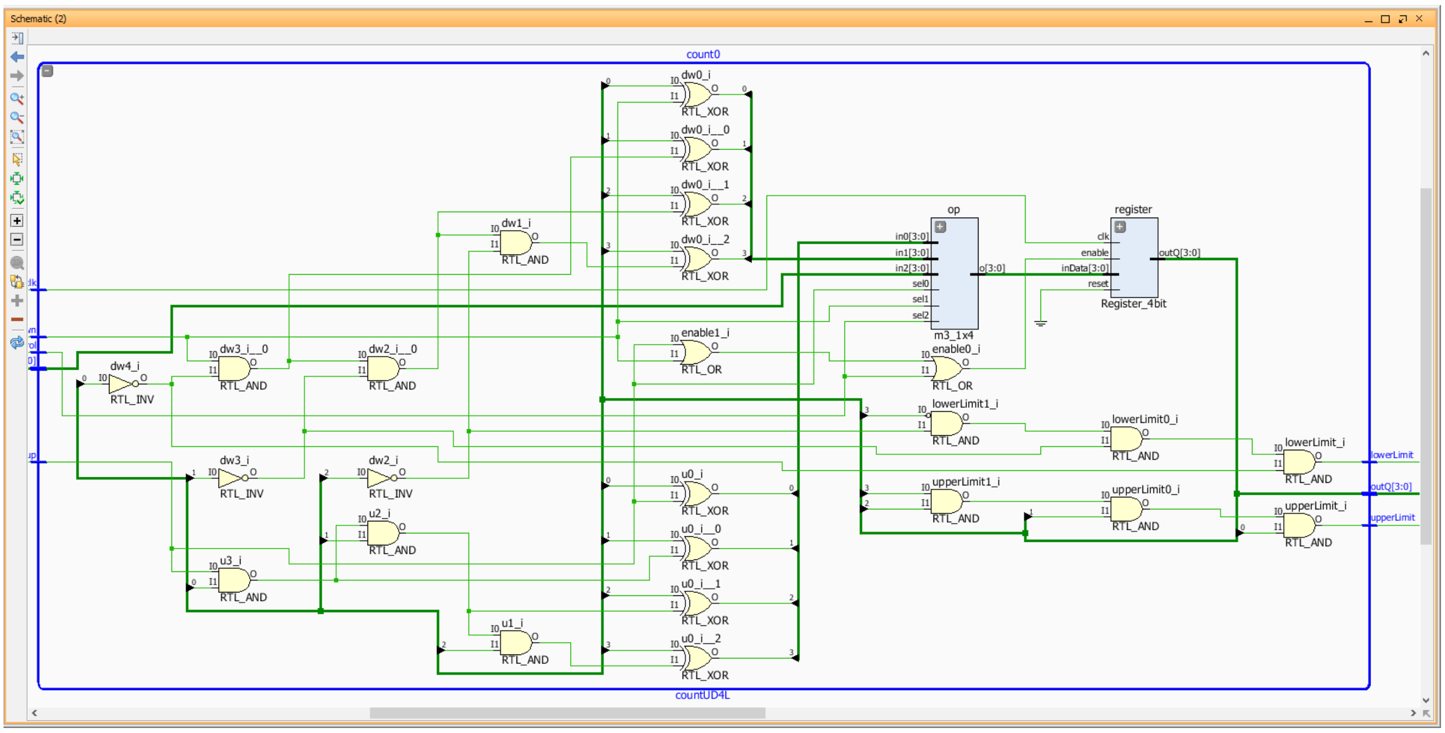
```
`timescale 1ns / 1ps
//UP REGISTER
module Register_4bit(
    input [3:0] inData,
    input clk,
    input enable,
    input reset,
    output [3:0] outQ
    );

    FDRE #(.INIT(1'b0) ) register0(.C(clk), .R(reset), .CE(enable), .D(inData[0]),
.Q(outQ[0]));
    FDRE #(.INIT(1'b0) ) register1(.C(clk), .R(reset), .CE(enable), .D(inData[1]),
.Q(outQ[1]));
    FDRE #(.INIT(1'b0) ) register2(.C(clk), .R(reset), .CE(enable), .D(inData[2]),
.Q(outQ[2]));
    FDRE #(.INIT(1'b0) ) register3(.C(clk), .R(reset), .CE(enable), .D(inData[3]),
.Q(outQ[3]));

endmodule
```

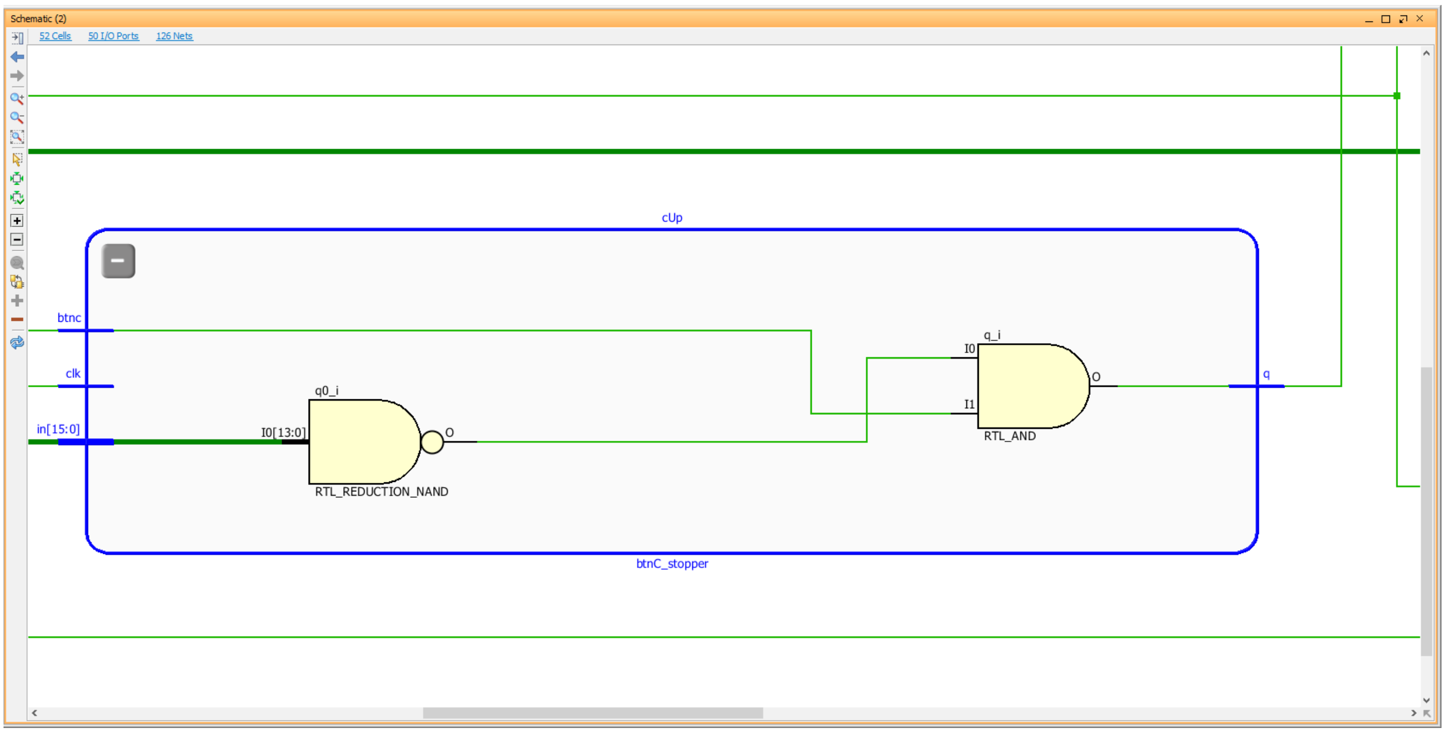
```
`timescale 1ns / 1ps

module countUD16L(
    input clk, up, down, loadControl,
    input [15:0] loadData,
    output upperLimit, lowerLimit,
    output [15:0] outQ
    );
    wire [3:0] p, q;

    assign lowerLimit = q[0] & q[1] & q[2] & q[3];
    assign upperLimit = p[0] & p[1] & p[2] & p[3];

    countUD4L count0(.clk(clk), .down(down),                      .lowerLimit(q[0]),
     .up(up),                      .upperLimit(p[0]),      .outQ(outQ[3:0]),
.loadData(loadData[3:0]),    .loadControl(loadControl));
    countUD4L count1(.clk(clk), .down(down & q[0]),               .lowerLimit(q[1]),
     .up(up & p[0]),               .upperLimit(p[1]),      .outQ(outQ[7:4]),
.loadData(loadData[7:4]),    .loadControl(loadControl));
    countUD4L count2(.clk(clk), .down(down & q[1] & q[0]),        .lowerLimit(q[2]),
     .up(up & p[1] & p[0]),        .upperLimit(p[2]),      .outQ(outQ[11:8]),
.loadData(loadData[11:8]),   .loadControl(loadControl) );
    countUD4L count3(.clk(clk), .down(down & q[2] & q[1] & q[0]), .lowerLimit(q[3]),
     .up(up & p[2] & p[1] & p[0]), .upperLimit(p[3]),      .outQ(outQ[15:12]),
.loadData(loadData[15:12]),  .loadControl(loadControl));

endmodule
```

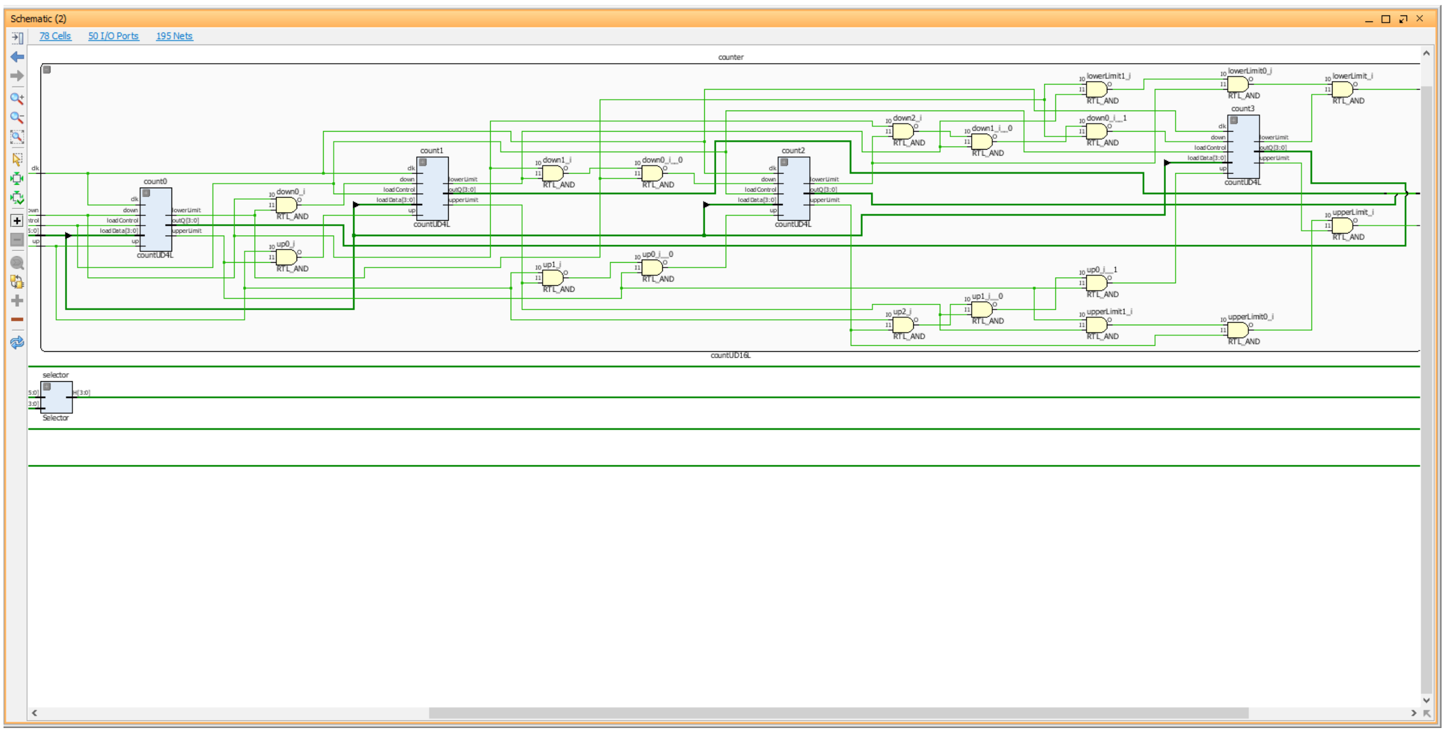

register
register0
clk
enable
inData[3:0]
reset
>C
CE
D
R
Q
FDRE
0
out(
register1
1
register2
2
register3
3
Register_4bit

```verilog
`timescale 1ns / 1ps

module edgeDetector(
    input in, clk,
    output edgeDetected
    );
    wire [2:0] p;
    FDRE #(.INIT(1'b0) ) register0(.C(clk), .R(1'b0), .CE(1'b1), .D(in), .Q(p[0]));
    FDRE #(.INIT(1'b0) ) register1(.C(clk), .R(1'b0), .CE(1'b1), .D(p[0]), .Q(p[1]));


    assign edgeDetected = in & ~p[0] & ~p[1];
endmodule
```

```
`timescale 1ns / 1ps


module btnC_stopper(
    input clk, btnc,
    input [15:0] in,
    output q

    );
    //wire i;
    // if in[15:2] = 1 output q = 0
    assign q = ~&{in[15:2]} & btnc;

      //  FDRE #(.INIT(1'b0) ) register0(.C(clk), .R(1'b0), .CE(1'b1), .D(i), .Q(q));

endmodule
```

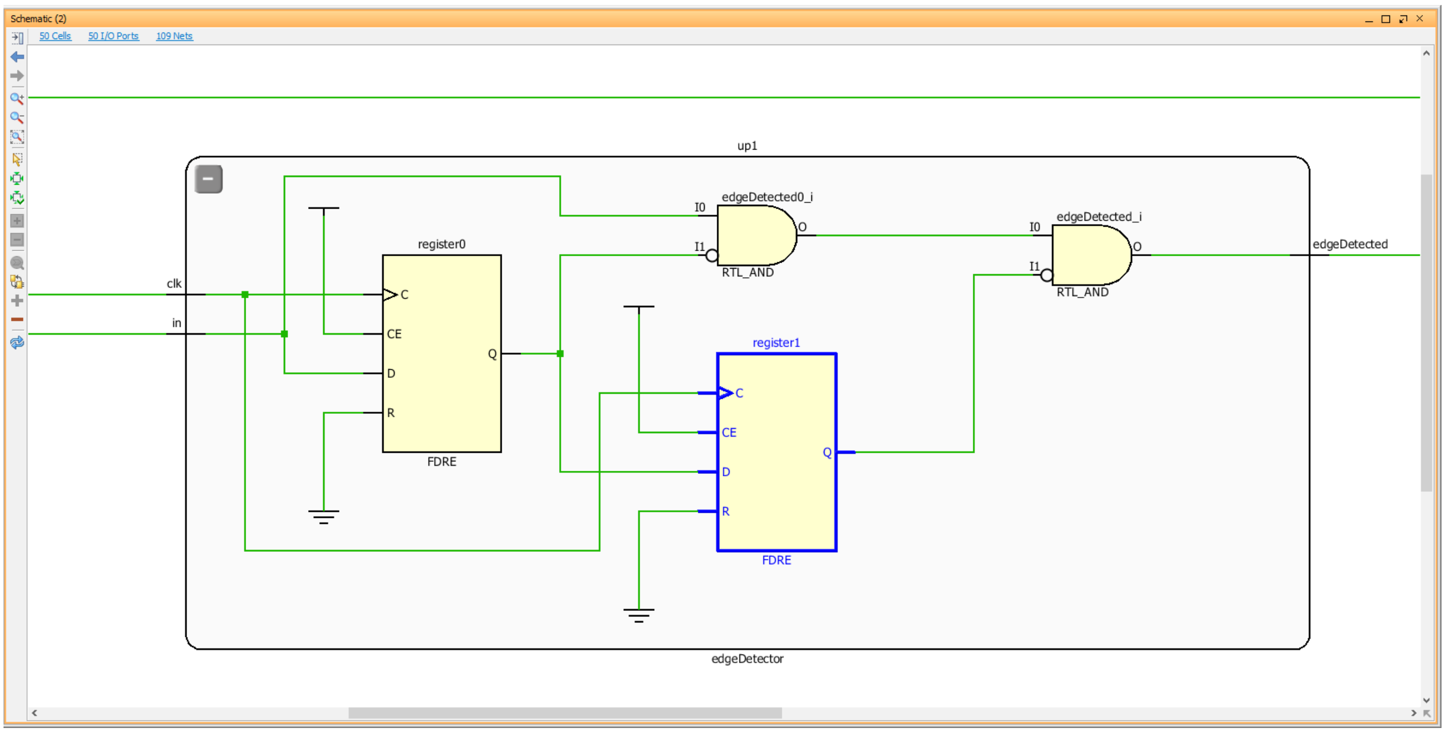

Schematic (2)
count0
countUD4L
dw0_i
RTL_XOR
dw0_i__0
RTL_XOR
dw0_i__1
RTL_XOR
dw0_i__2
RTL_XOR
dw1_i
RTL_AND
op
in0[3:0]
in1[3:0]
in2[3:0]
sel0
sel1
sel2
o[3:0]
m3_1x4
register
clk
enable
inData[3:0]
reset
outQ[3:0]
Register_4bit
enable1_i
RTL_OR
enable0_i
RTL_OR
dw3_i__0
RTL_AND
dw2_i__0
RTL_AND
dw4_i
RTL_INV
lowerLimit1_i
RTL_AND
lowerLimit0_i
RTL_AND
lowerLimit_i
RTL_AND
dw3_i
RTL_INV
dw2_i
RTL_INV
u0_i
RTL_XOR
upperLimit1_i
RTL_AND
upperLimit0_i
RTL_AND
upperLimit_i
RTL_AND
u2_i
RTL_AND
u0_i__0
RTL_XOR
u3_i
RTL_AND
u0_i__1
RTL_XOR
u1_i
RTL_AND
u0_i__2
RTL_XOR
lowerLimit
outQ[3:0]
upperLimit
clk
up

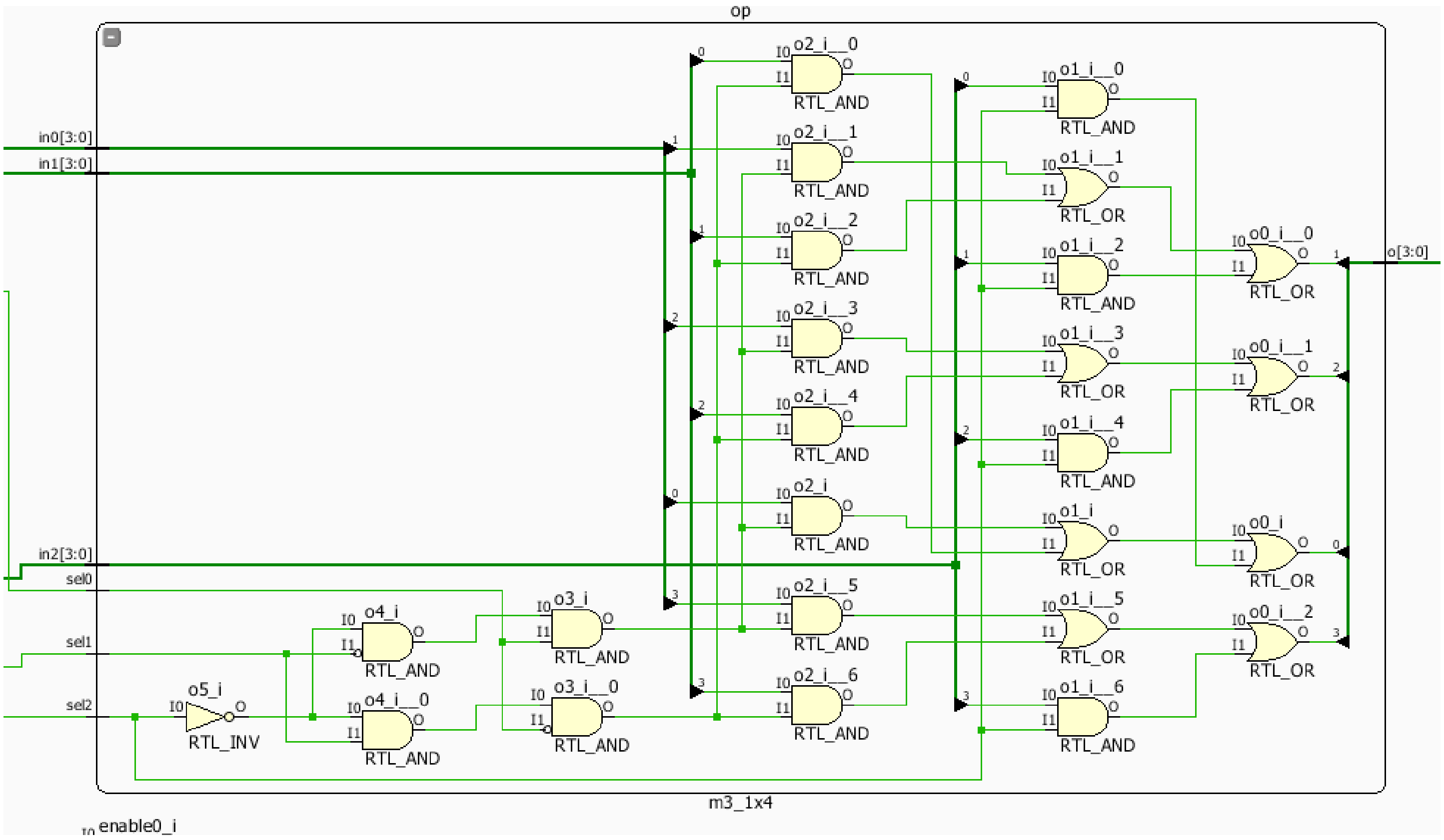

Schematic (2)
52 Cells
50 I/O Ports
126 Nets
cUp
btnc
clk
in[15:0]
q0_i
I0[13:0]
O
RTL_REDUCTION_NAND
q_i
I0
I1
O
RTL_AND
q
btnC_stopper

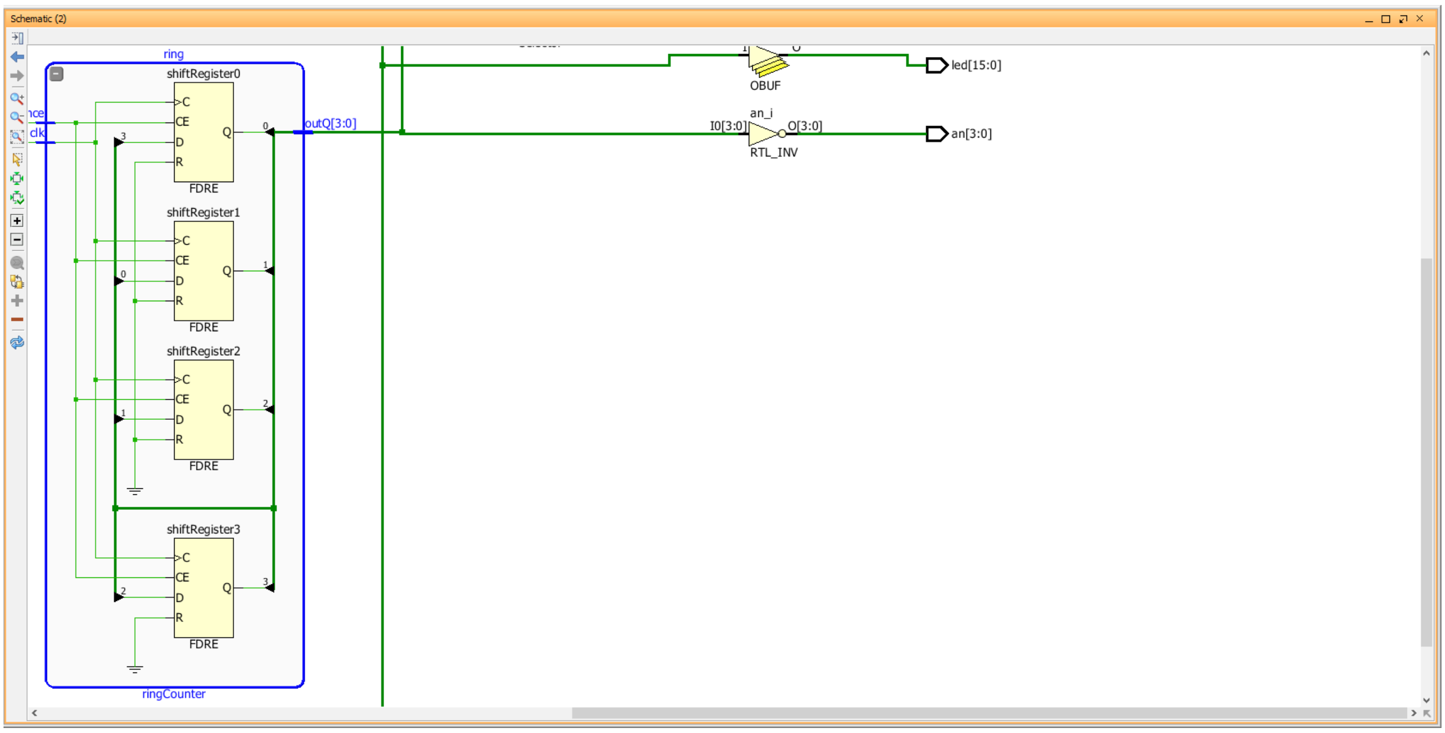

Schematic (2)
78 Cells
50 I/O Ports
195 Nets
counter
countUD16L
count0
count1
count2
count3
countUD4L
clk
down
load Control
load Data[3:0]
up
lowerLimit
outQ[3:0]
upperLimit
down0_i
up0_i
down1_i
down0_i__0
up1_i
up0_i__0
down2_i
down1_i__0
down0_i__1
lowerLimit1_i
lowerLimit0_i
lowerLimit_i
upperLimit_i
up0_i__1
up1_i__0
up2_i
upperLimit1_i
upperLimit0_i
RTL_AND
selector
Selector

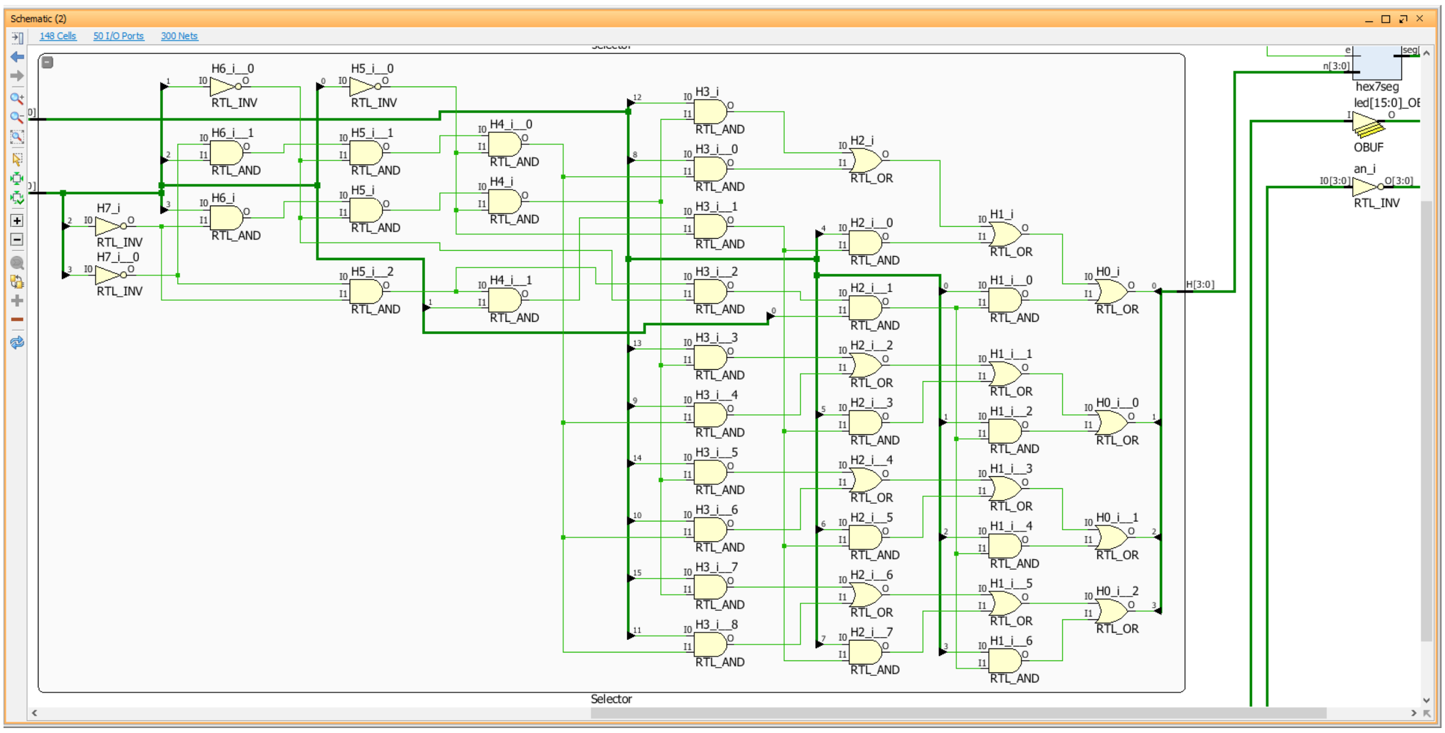
```
`timescale 1ns / 1ps

module m3_1x4(
    input [3:0] in0,
    input [3:0] in1,
    input [3:0] in2,
    input sel2, sel1, sel0,
    output [3:0] o
    );

     assign o[3] =  (in0[3] & (~sel2 & ~sel1 & sel0)) | (in1[3] & (~sel2 & sel1 &
~sel0)) | (in2[3] & (sel2));
     assign o[2] =  (in0[2] & (~sel2 & ~sel1 & sel0)) | (in1[2] & (~sel2 & sel1 &
~sel0)) | (in2[2] & (sel2));
     assign o[1] =  (in0[1] & (~sel2 & ~sel1 & sel0)) | (in1[1] & (~sel2 & sel1 &
~sel0)) | (in2[1] & (sel2));
     assign o[0] =  (in0[0] & (~sel2 & ~sel1 & sel0)) | (in1[0] & (~sel2 & sel1 &
~sel0)) | (in2[0] & (sel2));
endmodule
```

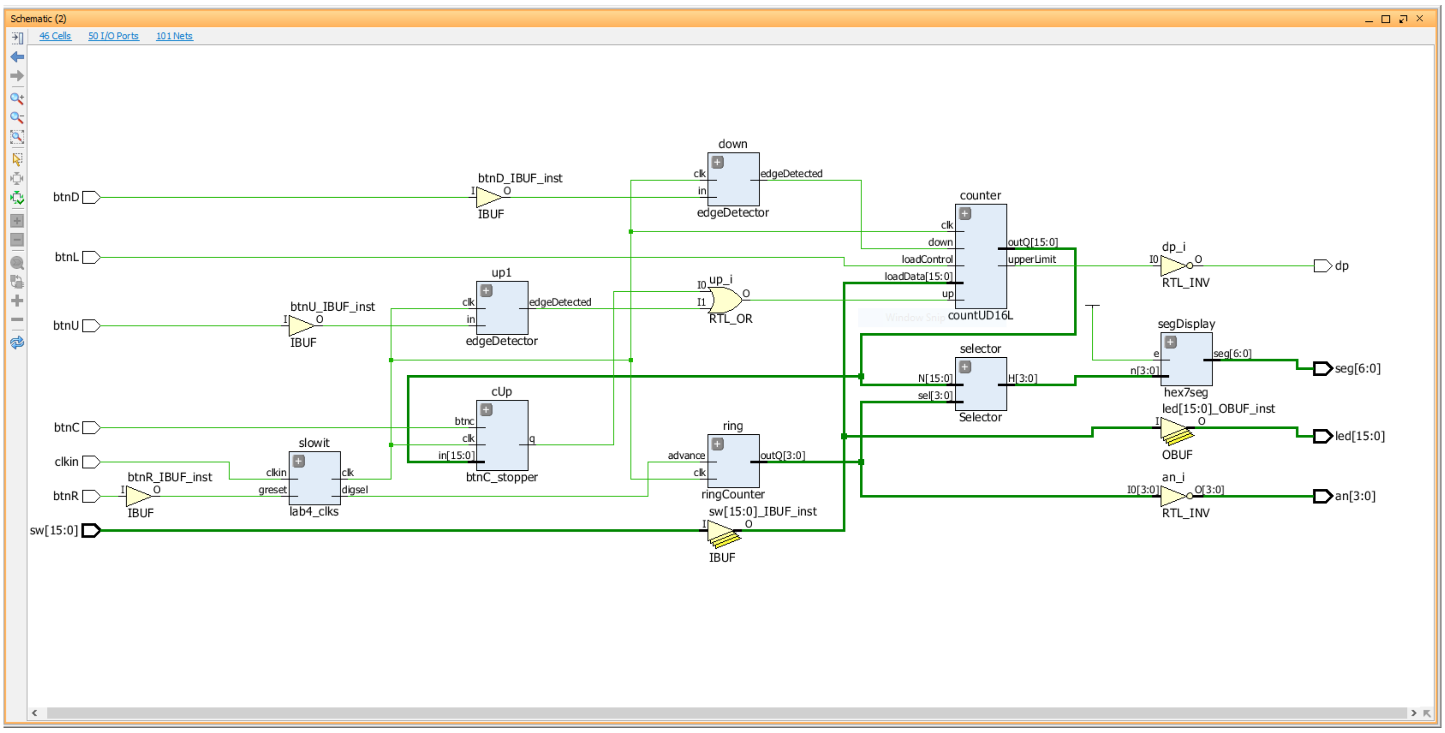
```
`timescale 1ns / 1ps

module ringCounter(
    input clk, advance,
    output [3:0] outQ
    );
    wire [3:0] p;
    assign outQ = p;
    FDRE #(.INIT(1'b1) ) shiftRegister0(.C(clk), .R(reset), .CE(advance), .D(p[3]),
.Q(p[0]));
    FDRE #(.INIT(1'b0) ) shiftRegister1(.C(clk), .R(reset), .CE(advance), .D(p[0]),
.Q(p[1]));
    FDRE #(.INIT(1'b0) ) shiftRegister2(.C(clk), .R(reset), .CE(advance), .D(p[1]),
.Q(p[2]));
    FDRE #(.INIT(1'b0) ) shiftRegister3(.C(clk), .R(reset), .CE(advance), .D(p[2]),
.Q(p[3]));


endmodule
```


Schematic (2)
50 Cells
50 I/O Ports
109 Nets
up1
edgeDetected0_i
I0
I1
O
RTL_AND
edgeDetected_i
I0
I1
O
RTL_AND
edgeDetected
register0
C
CE
D
R
Q
FDRE
register1
C
CE
D
R
Q
FDRE
clk
in
edgeDetector


op
in0[3:0]
in1[3:0]
in2[3:0]
sel0
sel1
sel2
o[3:0]
o2_i__0
o2_i__1
o2_i__2
o2_i__3
o2_i__4
o2_i
o2_i__5
o2_i__6
o1_i__0
o1_i__1
o1_i__2
o1_i__3
o1_i__4
o1_i
o1_i__5
o1_i__6
o0_i__0
o0_i__1
o0_i
o0_i__2
o3_i
o3_i__0
o4_i
o4_i__0
o5_i
RTL_AND
RTL_OR
RTL_INV
m3_1x4
enable0_i


Schematic (2)
ring
shiftRegister0
C
CE
D
R
Q
FDRE
shiftRegister1
FDRE
shiftRegister2
FDRE
shiftRegister3
FDRE
ringCounter
ce
clk
outQ[3:0]
OBUF
led[15:0]
an_i
I0[3:0]
O[3:0]
RTL_INV
an[3:0]


Schematic (2)
148 Cells
50 I/O Ports
300 Nets
Selector
H6_i__0
H5_i__0
H6_i__1
H5_i__1
H4_i__0
H6_i
H5_i
H4_i
H7_i
H7_i__0
H5_i__2
H4_i__1
H3_i
H3_i__0
H3_i__1
H3_i__2
H3_i__3
H3_i__4
H3_i__5
H3_i__6
H3_i__7
H3_i__8
H2_i
H2_i__0
H2_i__1
H2_i__2
H2_i__3
H2_i__4
H2_i__5
H2_i__6
H2_i__7
H1_i
H1_i__0
H1_i__1
H1_i__2
H1_i__3
H1_i__4
H1_i__5
H1_i__6
H0_i
H0_i__0
H0_i__1
H0_i__2
RTL_INV
RTL_AND
RTL_OR
H[3:0]
hex7seg
led[15:0]_OB
OBUF
an_i
RTL_INV
I0[3:0]
O[3:0]


Schematic (2)
46 Cells
50 I/O Ports
101 Nets
btnD
btnL
btnU
btnC
clkin
btnR
sw[15:0]
btnD_IBUF_inst
IBUF
btnU_IBUF_inst
IBUF
btnR_IBUF_inst
IBUF
down
edgeDetector
clk
in
edgeDetected
up1
edgeDetector
cUp
btnC_stopper
btnc
clk
in[15:0]
q
slowit
lab4_clks
clkin
greset
clk
digsel
up_i
RTL_OR
I0
I1
O
ring
ringCounter
advance
outQ[3:0]
sw[15:0]_IBUF_inst
IBUF
counter
countUD16L
down
loadControl
loadData[15:0]
up
outQ[15:0]
upperLimit
selector
Selector
N[15:0]
sel[3:0]
H[3:0]
dp_i
RTL_INV
segDisplay
hex7seg
e
n[3:0]
seg[6:0]
led[15:0]_OBUF_inst
OBUF
an_i
RTL_INV
I0[3:0]
O[3:0]
dp
seg[6:0]
led[15:0]
an[3:0]

```
`timescale 1ns / 1ps



module Selector(
    input [3:0] sel,
    input [15:0] N,
    output [3:0] H
    );

assign H[3] = (N[15] & ( sel[3] & ~sel[2] & ~sel[1] & ~sel[0]) ) | (N[11] & (~sel[3]
&  sel[2] & ~sel[1] & ~sel[0]) ) | (N[7] & (~sel[3] & ~sel[2] &  sel[1] & ~sel[0]) )
| (N[3] & (~sel[3] & ~sel[2] & ~sel[1] &  sel[0]) ) ;
assign H[2] = (N[14] & ( sel[3] & ~sel[2] & ~sel[1] & ~sel[0]) ) | (N[10] & (~sel[3]
&  sel[2] & ~sel[1] & ~sel[0]) ) | (N[6] & (~sel[3] & ~sel[2] &  sel[1] & ~sel[0]) )
| (N[2] & (~sel[3] & ~sel[2] & ~sel[1] &  sel[0]) ) ;
assign H[1] = (N[13] & ( sel[3] & ~sel[2] & ~sel[1] & ~sel[0]) ) | (N[9]  & (~sel[3]
&  sel[2] & ~sel[1] & ~sel[0]) ) | (N[5] & (~sel[3] & ~sel[2] &  sel[1] & ~sel[0]) )
| (N[1] & (~sel[3] & ~sel[2] & ~sel[1] &  sel[0]) ) ;
assign H[0] = (N[12] & ( sel[3] & ~sel[2] & ~sel[1] & ~sel[0]) ) | (N[8]  & (~sel[3]
&  sel[2] & ~sel[1] & ~sel[0]) ) | (N[4] & (~sel[3] & ~sel[2] &  sel[1] & ~sel[0]) )
| (N[0] & (~sel[3] & ~sel[2] & ~sel[1] &  sel[0]) ) ;
endmodule
```

```
`timescale 1ns / 1ps

module topModule(
    input clkin, btnR, btnU, btnD, btnL, btnC,
    input [15:0] sw,
    output [6:0] seg,
    output dp,
    output [3:0] an,
    output [15:0] led

    );

    wire btnuEdge, up ;
    wire [15:0] q; //16 bit counter output
    wire [3:0] r;  //ringCounter 4 bit output into selector
    wire [3:0] H;  //selector choosing 4 bits to send to display logic
    wire clk;
    wire digsel;
    wire utc;
    wire ltc;
    wire continuousUp;


    assign up = continuousUp | btnuEdge;
    assign led = sw;

    edgeDetector up1(.clk(clk), .in(btnU), .edgeDetected(btnuEdge));
    edgeDetector down(.clk(clk), .in(btnD), .edgeDetected(btnDEdge));
    btnC_stopper cUp(.clk(clk), .in(q), .btnc(btnC), .q(continuousUp));

    countUD16L counter(.clk(clk), .up(up), .down(btnDEdge), .loadControl(btnL),
.loadData(sw), .upperLimit(utc), .lowerLimit(ltc), .outQ(q));



    ringCounter ring(.clk(clk), .advance(digsel), .outQ(r));
    Selector selector(.sel(r), .N(q), .H(H));
    hex7seg segDisplay(.n(H), .e(1'b1), .seg(seg));

    lab4_clks slowit (.clkin(clkin), .greset(btnR), .clk(clk), .digsel(digsel));

    assign an = ~r;


    assign dp = ~(~an[3] & an[2] & an[1] & an[0] & utc | an[3] & an[2] & an[1] &
~an[0] & ltc);
endmodule
```